Hochschule Düsseldorf
University of Applied Sciences
HSD

# Vergleich von UVM-SystemC mit etablierten UVM-Implementierungen

Moritz Kupke, Stephan Gerth, Bernhard M. Rieß

***Zusammenfassung*—Ziel dieser Arbeit war es, die Anwendbarkeit von UVM-SystemC bei der Schaltungsverifikation zu überprüfen. Dazu wurde UVM-SystemC mit den beiden etablierten UVM-Implementierungen UVM-Specman *e* und UVM-SystemVerilog verglichen. Dieser Vergleich wurde anhand von zwei Designs, einem SFIFO und einem Arbiter, durchgeführt. Dazu wurden beide Schaltungen in allen drei UVM-Implementierungen verifiziert. Für die Verifikation wurden EDA-Tools der Firma Cadence Design Systems verwendet. Diese Arbeit kommt zu dem Ergebnis, dass UVM-SystemC im Allgemeinen anwendbar ist, jedoch einzelne Einschränkungen hingenommen werden müssen und stellenweise Verbesserungspotential besteht. Verbesserungspotenzial besteht unter anderem bei dem Tool Support durch die EDA-Tools. Positiv fallen die allgemeinen UVM Funktionen auf, hier ist UVM-SystemC vergleichbar mit den etablierten UVM-Implementierungen.**

***Schlüsselwörter*—Schaltungsverifikation, UVM, UVM-Specman *e*, UVM-SystemVerilog, UVM-SystemC**

## I. Einleitung

Die Verifikationsmethodik integrierter Schaltungen hat sich in den letzten Jahrzehnten stark gewandelt. Frühere Methoden sind aufgrund der gestiegenen Komplexität integrierter Schaltungen nicht mehr sinnvoll anwendbar, um eine umfassende Verifikation zu gewährleisten. Inzwischen hat sich die Universal Verification Methodology (UVM) als State-of-the-Art im Bereich Coverage-Driven-Verification durchgesetzt, um die Komplexität zu bewältigen. Diese bietet ein entsprechendes Verifikationsframework, um zusammen mit Constrained Randomization eine entsprechende Verifikationsabdeckung zu erreichen.

Die meistgenutzte Implementierung von UVM ist derzeit UVM-SystemVerilog, gefolgt von UVM-Specman *e*. Als dritte Implementierung wird derzeit UVM-SystemC von der Accellera SystemC Verification Working Group (VWG) [2] entwickelt und steht als öffentliche Proof-of-Concept Implementierung zur Verfügung. UVM-SystemC bietet als Alleinstellungsmerkmal die Unabhängigkeit von kommerziellen Simulatoren und somit eine größere Flexibilität im Austausch von Modellen mit Entwicklungspartnern und Kunden.

Inwieweit UVM-SystemC im Vergleich zu den etablierten UVM-Implementierungen anwendbar ist und welche Kriterien dafür berücksichtigt werden müssen wurde in dieser Arbeit untersucht. Die Kriterien für die Anwendbarkeit wurden aus den üblichen Implementierungen abgeleitet. Dazu zählen beispielsweise die Lizenzierung, die Weitergabe an Entwicklungspartner und Kunden, die Verwendung von Verifikations IPs, die Integration in bisherige Tools oder die Erkennbarkeit von Fehlern. Darüber hinaus wurde analysiert, in welchem Umfang bereits bestehende EDA-Tools für den Einsatz von UVM-SystemC genutzt werden können. Dazu wurden zwei Designs unterschiedlicher Komplexität in den drei untersuchten UVM-Implementierungen verifiziert. Anhand der beiden Designs konnten die Unterschiede zwischen den einzelnen UVM-Implementierungen herausgearbeitet werden. Abschließend wurden die vorab identifizierten Kriterien zur Bewertung der Anwendbarkeit von UVM-SystemC verwendet.

Moritz Kupke, moritz.kupke@bosch.com, Stephan Gerth, stephan.gerth@bosch.com, Bernhard Rieß, bernhard.riess@hs-duesseldorf.de

## II. Stand der Technik

UVM ist heutzutage eine etablierte Verifikationsmethodik, welche viele Vorteile bietet, wie z.B. die Wiederverwendbarkeit von Verifikationskomponenten. UVM wird bei vielen Verifikationsprozessen verwendet und profitiert davon, dass es unabhängig von einer Verifikationssprache ist. Dadurch stellt UVM sicher, dass die Entwicklungsergebnisse verschiedener Firmen kompatibel sind, egal welche Verifikationssprache Verwendung findet. Die Methodik wird heutzutage von drei verschiedenen Verifikationssprachen unterstützt: Specman *e*, SystemVerilog und SystemC. Da UVM bei allen Implementierungen Gemeinsamkeiten aufweist, werden diese zunächst im Allgemeinen erläutert und im späteren Teil wird auf die einzelnen Sprachen eingegangen.

### A. Aufbau einer UVM-Umgebung

Zur Schaltungsverifikation werden zwei Komponenten benötigt: Das Device under Verification (DUV) und die Verifikationsumgebung (siehe Abbildung 1). Diese werden in der Testbench (TB) verbunden. Die Verbindung zwischen den beiden Komponenten wird über ein sogenanntes Interface (IF) oder eine Signal Map (SMP) realisiert. [1]

Die oberste Ebene der Verifikationsumgebung trägt den Namen Environment (ENV) und bildet das Top

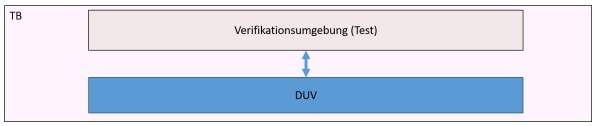



Abbildung 1. Testbench

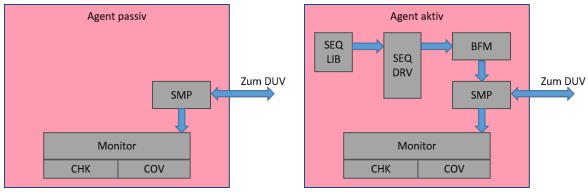



Abbildung 2. passiver Agent li., aktiver Agent re.

Modul der Verifikation. Das ENV instanziiert mindestens einen Agent, welcher ein passiver oder ein aktiver Agent sein kann.
Ein passiver Agent (Abbildung 2 links) übernimmt eine aufzeichnende Funktion und besteht deshalb nur aus einer Komponente, dem Monitor.

Der Monitor greift alle Signale ab, die an einem bestimmten Interface anliegen (blaue Pfeile in Abbildung 2 links) und zeichnet die Kommunikation auf. Zudem kann der Monitor die Coverage aufzeichnen. Coverage bedeutet, dass überprüft wird, ob z.B. ein Signal bestimmte Werte annimmt oder nicht. Man unterscheidet zwischen der Toggle-Coverage (geht ein Signal von 0 auf 1 und umgekehrt), der Finite State Machine (FSM)-Coverage (werden alle Zustände einer Schaltung erreicht), der Code-Coverage (wird jede Zeile mindestens einmal ausgeführt) und der Assertion-Coverage (werden alle Assertions tatsächlich überprüft). Zudem ist es möglich, Checks für die Überprüfung bestimmter Eigenschaften, die das DUV erfüllen muss, aufzurufen.

Ein aktiver Agent (Abbildung 2 rechts) hingegen ergänzt den passiven Agent um einen Sequence Driver (SEQ-DRV), eine Sequence Driver Library (SEQ-LIB) und ein Bus Functional Model (BFM).

Aufgabe des SEQ-DRV ist es Test-Sequenzen (Test-Pattern) zu erzeugen, die dann über das BFM an die Signal Map gelangen (blaue Pfeile in Abbildung 2 rechts) und das DUV stimulieren. Die Test-Sequenzen werden in einem zusätzlichen Modul, der sogenannten Sequence Driver Library (SEQ-LIB), erstellt, gespeichert und vom SEQ-DRV aufgerufen. Zusätzlich zur SEQ-LIB ist es möglich ein Sequenz Item zu verwenden, welches alle relevanten Variablen instanziiert.

Neben dem Agent kann die ENV ein Scoreboard (SCRBRD) zur Überprüfung einbinden wie in Abbildung 3 dargestellt. Dessen Aufgabe besteht darin, die Daten aus den Monitoren zu vergleichen. Empfängt das Scoreboard von der Spezifikation abweichende Daten, wird eine Fehlermeldung angezeigt. Die Daten des Monitors werden über den sogenannten Transaction

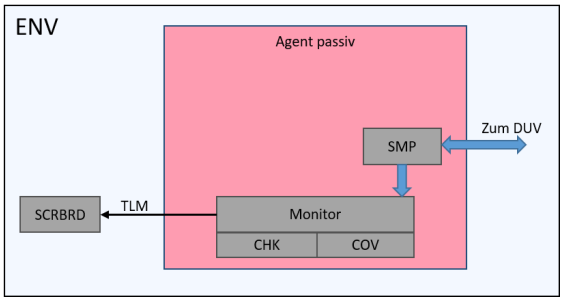



Abbildung 3. UVM ENV mit Agent und Scoreboard

Level Modeling Port (TLM-Port) übertragen (schwarze Pfeile in Abbildung 3). Der TLM-Port eignet sich für die Kommunikation zwischen einzelnen Modulen. Die gleichzeitige Verbindung zwischen mehreren Modulen ist möglich.

Zusätzlich ermittelt das Scoreboard eine Coverage, um weitere Fehler zu erkennen bzw. die Testabdeckung zu überprüfen. Zusätzlich zu Scoreboard und Agent kann die ENV weitere ENV instanziieren wie in Abbildung 4 dargestellt.

Für die automatisierte Erstellung von Komponenten wie z.B. des Agents kann eine Konfiguration (Config) verwendet werden. Diese erstellt dem Entwickler wiederkehrende Codestrukturen und spart somit Zeit. Gibt es in der Testbench mehrere Environments oder Agents, so muss zusätzlich gewährleistet werden, dass alle SEQ-DRV synchron arbeiten [1]. Diese Synchronisierung übernimmt der Virtual Sequence Driver (VSEQ). Der VSEQ ist mit allen untergeordneten Sequence Drivern verbunden (rote Pfeile in Abbildung 4) und kann in jedem verbundenem SEQ-DRV einzelne Sequenzen starten. Um mehrere verschiedene Szenarien (Eingangsfolgen am DUV) zu erstellen, kann sich der VSEQ aus der Virtual Sequence Driver Library (VSEQ-LIB) bedienen. Dort sind alle möglichen Verknüpfungen der jeweiligen Sequence Driver hinterlegt.

Damit verschiedene Test-Szenarien erstellt werden können, gibt es die letzte Hierarchieebene, den Testcase (Test in Abbildung 4). Die Aufgabe des Testcases ist es, verschiedene VSEQ zu verknüpfen und zu starten. Zusätzlich legt der Testcase Start und Ende des Testes fest. Zur Verdeutlichung, dass eine ENV weitere Environments instanziieren kann, wurden in Abbildung 4 exemplarisch zwei Environments eingebunden.

### B. *UVM-Specman e*

UVM-Specman *e* ist eine Hardwareverifikationssprache (HVL), welche 1992 von Yoav Hollander ins Leben gerufen wurde. Weiterentwickelt wurde sie von der Firma Verisity, welche später von der Firma Cadence Design Systems (Cadence) aufgekauft wurde. UVM-Specman *e* zeichnet sich dadurch aus, dass sie speziell für die Verifikation entwickelt wurde. Das heißt mit der Sprache kann kein Design modelliert

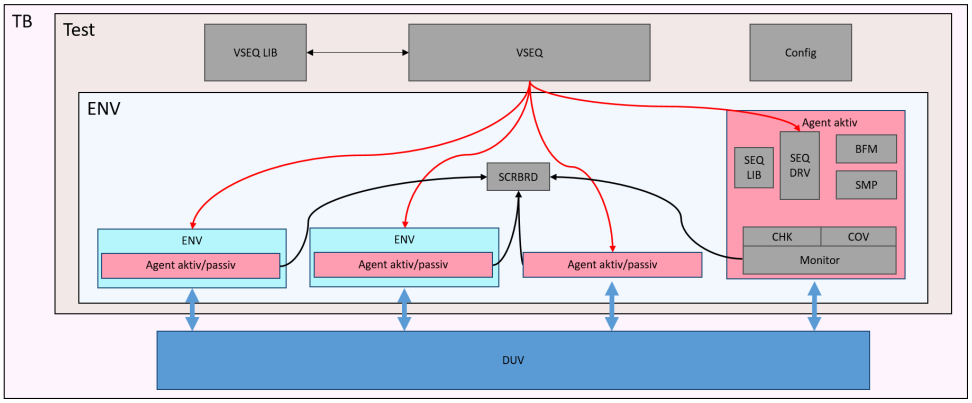



Abbildung 4. UVM Umgebung

werden, jedoch kann die Verifikation von Designs in allen gängigen Hardwarebeschreibungssprachen, wie z.B. Verilog, VHDL oder SystemC, erfolgen. *e* ist eine aspektorientierte Programmiersprache (AOP). Damit bei dieser Sprache die Effizienz und Wiederverwendbarkeit gegeben ist, wurde für sie die *e*RM (*e* Reuse Methodology) eingeführt. Diese soll zudem die Verifikationszeiten, durch u.a. den Re-use von Komponenten, verkürzen. Mit der Gründung von Accellera wurde dann die UVM-Methodik für *e* adaptiert und bis heute verwendet.

### C. UVM-SystemVerilog

Im Gegensatz zu UVM-Specman *e* ist UVM-SystemVerilog keine reine HVL, denn bei UVM-SystemVerilog handelt es sich um eine Weiterentwicklung der Hardwarebeschreibungssprache (HDL) Verilog (IEEE 1364). UVM-SystemVerilog wurde von Accellera entwickelt und in der IEEE 1800 standardisiert. Dank der Entwicklung durch Accellera ist UVM-SystemVerilog eine frei verfügbare Sprache und es gibt frei verfügbare Dokumentationen und Beispiele [1]. Im Gegensatz z.B. zu UVM-Specman *e* ist UVM-SystemVerilog nicht an kommerzielle Simulatoren gebunden und darüber hinaus eine objektorientierte Programmiersprache (OOP). Wie auch UVM-Specman *e* unterstützt UVM-SystemVerilog die UVM Methodik. Bei UVM-SystemVerilog ist UVM als Klassenbibliothek eingebunden und kann so Verwendung finden.

### D. UVM-SystemC

UVM-SystemC ist die neueste UVM-Implementierung von Accellera und in der IEEE 1666 standardisiert [2]. UVM-SystemC ist keine reine HVL, sondern ergänzt die SystemC HDL Implementierung. Wie für UVM-SystemVerilog wurde für UVM-SystemC eine UVM-Klassen Bibliothek angelegt. UVM-SystemC ist zudem keine eigene „Programmiersprache“, sondern eine Erweiterung der objektorientierten Sprache C++. Dies bietet den Vorteil, dass keine kommerziellen Simulatoren verwendet werden müssen sondern ein frei verfügbarer C++-Compiler genügt. Weiterhin kann die Beschreibung in einer höheren Abstraktionsebene erfolgen, wie z.B. durch Verwendung von TLM Ports. Durch viele verfügbaren Dokumentationen und Beispiele lässt sich die Verifikationsumgebung zudem leicht adaptieren. Hinzu kommt, dass UVM-SystemC DUVs in allen gängigen HDLs unterstützt.

## III. Vergleich und Anwendung der UVM-Implementierungen

Für den Vergleich der drei verschiedenen UVM-Implementierungen wurden zunächst zwei repräsentative Designs indentifiziert, die im Folgenden kurz charakterisiert werden. Anschließend wurden die anzuwendenden Vergleichskriterien definiert und mit Hilfe einer Entscheidungsmatrix untereinander priorisiert. Dann wurde ein Verifikationsplan entwickelt um einen einheitlichen Vergleich zwischen den drei UVM-Implementierungen zu gewährleisten und damit die einzelnen Vor- und Nachteile herausarbeiten zu können. Dazu wird auch die verwendete Entwicklungsumgebung kurz beschrieben. Abschließend werden die Ergebnisse des Vergleichs diskutiert.

### A. Testcases

Im Rahmen dieser Arbeit wurden zwei unterschiedliche Designs mit den drei UVM-Implementierungen verifiziert. Die beiden Designs lagen jeweils in der Beschreibungssprache VHDL vor und wurden in allen drei UVM-Implementierungen in VHDL verifiziert.

Als erstes Modul wurde ein **SFIFO** (Synchronous First In First Out Buffer) gewählt (vgl. Abbildung 5. Das SFIFO ist ein einfaches DUV und sollte einen schnellen und einfachen Überblick über die Anwendbarkeit von UVM-SystemC geben. Es verfügt über 9 Ein- und Ausgänge. Diese lassen sich in drei funktionale Einheiten unterteilen. Die erste Einheit (FIFO Main) stellt das Taktsignal (clk) und den asynchronen Reset (reset_n). Die zweite Einheit (WRITE) ist für das Schreiben von Daten in das FIFO zuständig und setzt

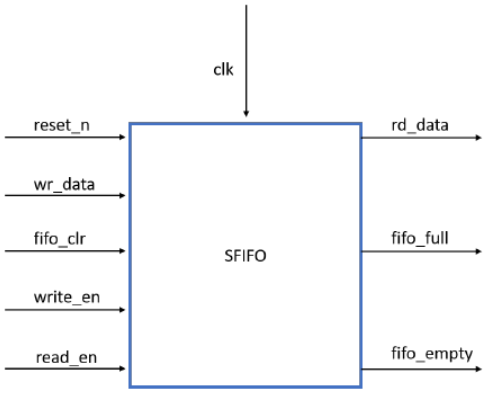



Abbildung 5. SFIFO

sich aus den Logik-Signalen write enable (write_en), write Data (wr_data) und fifo full (fifo_full) zusammen. Ist das write_en Signal auf high gesetzt und liegt am Taktsignal eine steigende Taktflanke an, so werden die Daten über die wr_data Bus Leitung in den FIFO geschrieben. Die Bus Leitung kann eine variable Breite von 1 bis 256 Bit haben. Über das Signal fifo_full wird dem Anwender signalisiert, ob der FIFO voll ist (das Signal ist high) oder nicht (das Signal ist low). Die Änderung des Signals wird immer mit der nächsten steigenden Taktflanke ausgegeben. Zusätzlich ist die Anzahl der Speicherplätze fest einstellbar. Als dritte Einheit (READ) gibt es die Lese-Einheit, die für das Auslesen des FIFOs zuständig ist. Dafür signalisiert ein read enable Signal (read_en) bei einem positiven Wert und einer steigenden Taktflanke dem FIFO, dass Daten ausgelesen werden sollen. Die Daten liegen mit der nächsten steigenden Taktflanke an der read Datenleitung (rd_data) an. Die Datenleitung hat die selbe Breite wie die wr_data Leitung. Damit der Anwender informiert wird, wenn der FIFO leer ist, also keine Daten mehr im Speicher sind, gibt es das fifo_empty Signal welches auf high geht, wenn der FIFO leer ist.

Als zweites Modul wurde ein **Arbiter** verwendet. Der Arbiter ist ein komplexeres und aufwendigeres Modul als der SFIFO. Er verfügt über 3 APB-Masterinterfaces und 4 APB-Slaveinterfaces, sowie einer Logik zum Verteilen der Zugriffe der Masters auf die Slaves. Der Arbiter soll die Kommunikation zwischen drei Master-Modulen und vier Slave-Modulen steuern. Die Kommunikation ist bidirektional d.h., es ist möglich Daten vom Master zum Slave zu senden und Daten vom Slave zum Master zu übertragen. Alle Module kommunizieren nach dem AMBA 3 APB Protokoll von ARM Limited. Die Module sind nach folgendem Schema verbunden, wie in Abbildung 6 zu sehen:

Master 0 kann mit Slave 0 und 1 kommunizieren (blaue Pfeile). Master 1 ist mit Slave 0, 1 und 2 verbunden (grüne Pfeile). Master 3 kommuniziert mit Slave 1, 2 und 3 (rosa Pfeile). Hierbei muss beachtet

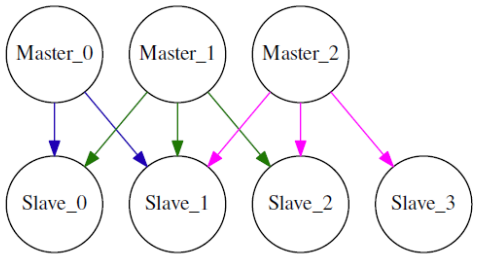



Abbildung 6. Arbiter.

werden, dass die Master-Module gleichzeitig kommunizieren können. Jedoch darf es nicht passieren, dass beispielsweise Master 2 und Master 1 gleichzeitig auf Slave 2 zugreifen. Dafür ist eine Priorisierung in der Schaltung vorgesehen, dies bedeutet, dass der Master mit der höheren Priorität Vorrang erhält. In der konkreten Anwendung soll Master 0 die höchste Priorität bekommen und Master 2 die geringste Priorität. In dem vorherigen Beispiel würde dies bedeuten, das Master 1 den Vorrang erhält und Master 2 warten muss.

Der Arbiter bot den Vorteil, dass komplexere Probleme, wie z.B. die Verwendung der Konfigurierbarkeit oder des UVM-Debuggings, intensiv analysiert werden konnten.

### *B. Vergleichskriterien*

Für den Vergleich der drei UVM-Implementierungen wurde eine Vielzahl von Kriterien herausgearbeitet, die im Folgenden beschrieben werden. Sie finden dann im weiteren Vergleich auf beide Testcases Anwendung.

- **DUV-Language**
  Beschreibt in welchen Sprachen DUVs modelliert sein können, um mit der jeweiligen UVM-Implementierung verifiziert werden zu können.
- **TLM-Kommunikation**
  Beschreibt die Kommunikation zwischen einzelnen Modulen. Wie z.B. zwischen dem Monitor und dem Scoreboard.
- **Konfigurierbarkeit**
  Ist ein Modul über bestimmte Variablen anpassbar?
- **Constrained Randomization**
  Wie funktioniert die Randomisierung bzw. die Einschränkung der Randomisierung?
- **Coverage**
  Wie kann die Coverage erzeugt und gespeichert werden? Gibt es Unterschiede beim Erstellen der Coverage?
- **Checks**
  Können Checks verwendet werden?
- **Lines of Code (LOC)**
  Wie viele Zeilen Programmcode im Simulationsskript sind nötig?

- **UVM-Methoden**
  Werden alle UVM-Methoden bzw. -Funktionen unterstützt?
- **Templates**
  Können Templates zur Erstellung der Verifikationsumgebung verwendet werden?
- **Wiederverwendung (Re-use)**
  Welche Komponenten können wiederverwendet werden? An welchen Stellen im Verifikationscode müssen Änderungen vorgenommen werden?
- **Verification IPs (VIPs)**
  Gibt es wiederverwendbare Verifikations IPs? Welche Hersteller stellen VIPs zur Verfügung?
- **Weitergabe an den Kunden**
  Ist es möglich die fertige Testbench in einer Form an Entwicklungspartner und Kunden weiterzugeben, so dass sie dort gut genutzt werden kann?
- **EDA-Tool-Support**
  Gibt es Unterstützung von EDA-Tools? Hier wurde die Unterstützung für die UVM-Implementierungen mit Tools der Fa. Cadence geprüft.
- **Code Debugging im EDA-Tool**
  Wie lassen sich Fehler im DUV finden?
- **UVM-Debugging**
  Wie können einzelne UVM-Methoden im Simulator angezeigt werden?
- **Anzeigen bzw. Ausblenden von Konsolenausgaben**
  Können Info- oder Fehlermeldungen angezeigt oder ausgeblendet werden?
- **Lizenzkosten**
  Für welche der eingesetzten Tools fallen Lizenzkosten an?
- **Simulationszeit**
  Beschreibt die Zeit, die für die Ausführung der Simulation am Rechner benötigt wird.
- **UVM-Multi-Language (UVM-ML)**
  Ist es möglich DUVs, die in verschiedenen Hardwarebeschreibungssprachen modelliert sind, zu verifizieren?
- **Regression**
  Beschreibt das Handling von mehreren Testcases, sowie die Auswertung.
- **Ein- und Ausblenden von Coverpunkten**
  Kann die Coverage-Anzeige im Nachhinein verändert werden?

In Summe wurden somit 21 Kriterien erarbeitet, anhand derer die Anwendbarkeit von UVM-SystemC im Vergleich zu UVM-Specman *e* und UVM-SystemVerilog im Folgenden praktisch analysiert werden soll.

### C. Erstellung einer Entscheidungsmatrix

Um eine Prioritätsreihenfolge der einzelnen Vergleichskriterien festzulegen zu können wurde eine Entscheidungsmatrix entwickelt. In dieser Entscheidungsmatrix wurde die Priorität jedes einzelnen Kriteriums mit jedem anderen Kriterium verglichen und daraus eine Gesamtpunktzahl für jedes einzelne Kriterium ermittelt, welche dessen Bedeutung und damit Priorität definiert. Dazu wurden alle Kriterien in die Zeilen und Spalten der Entscheidungsmatrix eingetragen wie beispielhaft in Tabelle I dargestellt.

Tabelle I
BEISPIEL ENTSCHEIDUNGSMATRIX.

| Kriterium | A | B | C | Gesamt |
|---|---|---|---|---|
| A | x | 2 | 1 | 3 |
| B | 0 | x | 0 | 0 |
| C | 1 | 2 | x | 3 |

In diesem Beispiel soll Kriterium A wichtiger sein als Kriterium B, deshalb wird in die entsprechende Zelle (Zeile A, Spalte B) eine 2 eingetragen. Weiterhin sind A und C gleich wichtig, daher wird in die Zelle (A, C) eine 1 eingetragen. Ist B weniger wichtig als C, wird in Zelle (B, C) eine 0 eingetragen. Die Zellen unterhalb der Diagonale der Matrix ergeben sich durch das „Umkehren“ der Werte oberhalb der Diagonale. So wird beispielsweise der Wert für die Zelle (B, A) berechnet durch $W_{B,A} = 2 - W_{A,B} = 2 - 2 = 0$.

Die Entscheidungen in Tabelle II wurden für alle 21 Kriterien nach der allgemeinen Auffassung durchgeführt und sollten für die meisten Anwendungsfälle gültig sein. Falls der Anwender eigene Gewichtungen bzw. Entscheidungen vornehmen möchte, kann dies natürlich in einer modifizierten Tabelle berücksichtigt werden. Sind alle Entscheidungen durchgeführt, ergibt sich am Ende durch die Addition der Werte in allen Spalten einer Zeile die Gesamtpunktzahl für jedes einzelne Kriterium. Die Gesamtpunktzahl wurde dann als Indikator für die Priorität der einzelnen Kriterien verwendet. Aus der Matrix kann abgeleitet werden, dass die Lizenzkosten mit 40 Punkten die höchste Punktzahl erreichen und somit das wichtigste Kriterium darstellen. Die niedrigste Punktzahl (3) erreicht die Weitergabe an den Kunden.

Basierend auf den mit Hilfe dieser Entscheidungsmatrix ermittelten Gesamtpunktzahlen für jedes Kriterium werden dann in Kapitel IV der Ergebnisse des Vergleichs der drei UVM-Implementierungen bewertet und verglichen.

### D. Verifikationsplanung und -durchführung

Um die einzelnen Unterschiede und Gemeinsamkeiten der drei UVM-Implementierungen praxisnah vergleichen zu können wurde ein Verifikationsplan erstellt und abgearbeitet. Für eine erfolgreiche Verifikation eines DUV hat sich der folgende sechsschrittige Prozess etabliert:

**1. Erstellen eines Verifikationsplans (VPlan):**
Der VPlan gliedert sich in drei Einheiten. Die erste

Tabelle II
ENTSCHEIDUNGSMATRIX

| Kriterien | DUV Language | TLM-Kommunikation | Konfigurierbarkeit | Randomisierung | Coverage | Checks | LOC | UVM-Methoden | Templates | Wiederverwendung | Verification IPs | Kunden | EDA-Tool-Support | Code im EDA-Tool | UVM-Debugging | Konsolenausgaben | Lizenzkosten | Simulationszeit | UVM-Multi-Language | Regression | Coverage-Anzeige | **Gesamtpunktzahl** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DUVLanguage | x | 1 | 2 | 2 | 2 | 2 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | **18** |
| TLM-Kommunikation | 1 | x | 2 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 1 | 1 | 1 | 2 | 0 | 1 | 1 | 1 | 2 | **25** |
| Konfigurierbarkeit | 0 | 0 | x | 0 | 0 | 0 | 1 | 1 | 2 | 1 | 1 | 2 | 1 | 2 | 2 | 1 | 0 | 2 | 1 | 2 | 0 | **19** |
| Randomisierung | 0 | 1 | 2 | x | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 2 | 2 | 1 | 2 | **32** |
| Coverage | 0 | 1 | 2 | 1 | x | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 2 | 2 | 1 | 1 | **32** |
| Checks | 0 | 1 | 2 | 1 | 0 | x | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 2 | 2 | 1 | 1 | **30** |
| LOC | 1 | 0 | 1 | 0 | 0 | 0 | x | 1 | 1 | 0 | 1 | 2 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | **10** |
| UVM-Methoden | 2 | 1 | 1 | 0 | 0 | 0 | 1 | x | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 0 | 0 | 2 | 0 | 0 | **16** |
| Templates | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | x | 0 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | **8** |
| Wiederverwendung (Re-use) | 2 | 1 | 1 | 0 | 0 | 0 | 2 | 1 | 2 | x | 2 | 2 | 1 | 1 | 1 | 2 | 0 | 1 | 2 | 1 | 2 | **24** |
| Verification IPs | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | x | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | **7** |
| Weitergabe an Kunden | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **3** |
| EDA-Tool-Support | 1 | 1 | 1 | 0 | 0 | 0 | 2 | 1 | 2 | 1 | 2 | 2 | x | 1 | 2 | 1 | 0 | 0 | 2 | 1 | 1 | **21** |
| Code im EDA-Tool | 2 | 1 | 0 | 0 | 0 | 0 | 2 | 1 | 2 | 1 | 2 | 2 | 1 | x | 1 | 2 | 0 | 2 | 2 | 1 | 2 | **24** |
| UVM-Debugging | 2 | 1 | 0 | 0 | 0 | 0 | 2 | 1 | 2 | 1 | 2 | 2 | 0 | 1 | x | 2 | 0 | 1 | 2 | 1 | 1 | **21** |
| Konsolenausgaben | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 2 | 0 | 2 | 2 | 1 | 0 | 0 | x | 0 | 0 | 2 | 0 | 0 | **13** |
| Lizenzkosten | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | x | 2 | 2 | 2 | 2 | **40** |
| Simulationszeit | 2 | 1 | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 1 | 2 | 2 | 2 | 0 | 1 | 2 | 0 | x | 2 | 1 | 2 | **23** |
| UVM-Multi-Language | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | x | 0 | 0 | **7** |
| Regression | 1 | 1 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 2 | 2 | 1 | 1 | 1 | 2 | 0 | 1 | 2 | x | 2 | **26** |
| Coverage-Anzeige | 1 | 0 | 2 | 0 | 1 | 1 | 2 | 2 | 2 | 0 | 2 | 2 | 1 | 0 | 1 | 2 | 0 | 0 | 2 | 0 | x | **21** |

Einheit legt fest, welche Ein- und Ausgangssignale eines DUV gecovert werden sollen (Coverage). die zweite Einheit definiert, welche Szenarien überprüft werden sollen (Checks). Dazu wird die DUV-Spezifikation herangezogen. Die letzte Einheit ist die Code-Coverage. Zur Implementierung von Verifikationsplänen gibt es Tools der EDA-Hersteller, um diese am Ende des Prozesses analysieren und in der Regression wiederverwenden zu können.

**2. Erstellen der Verifikationsumgebung:**
Ist der VPlan abgeschlossen, folgt die konzeptionelle Erstellung der Verifikationsumgebung. Diese wird zunächst graphisch entwickelt und sollte alle benötigten Komponenten, wie in Kapitel II beschrieben enthalten. Hierbei ist es empfehlenswert, die einzelnen Environments und Agents in Funktionsgruppen zu gliedern.

**3. Implementierung der Verifikationskomponenten:**
Nach Fertigstellung der konzeptionellen Verifikationsumgebung kann mit der Implementierung begonnen werden. Die größten Unterschiede zwischen den drei UVM-Methodiken treten bei ihrer Implementierung auf. Es wird empfohlen für jede UVM-Komponente eine eigene Datei anzulegen und mittels geeigneter Ordner zu strukturieren.
Mit Hilfe des DUV SFIFO wurde analysiert, welche Templates zur Verfügung stehen. Insbesondere sollten hier die SEQ-LIB mit der Randomisierung, das Scoreboard mit der TLM-Kommunikation zu den Monitoren, die Coverage, die Checks und die Konfigurierbarkeit überprüft werden. In diesem Zusammenhang ist es sinnvoll zu erörtern, welche Komponenten wiederverwendet werden können.
Um einen weiteren Überblick über die Anzahl der code-relevanten Zeilen (Lines of Code (LOC)) zu bekommen, zählt das Tool `Cloc` [8] die Anzahl der wirksamen Codezeilen aber nicht die Kommentare oder die Länge der Zeilen.
Die Testbench ruft die Testcases auf. Dazu floß der Aufruf von Sequenzen in den Vergleich ein. Zudem wurde verglichen, wie der Aufruf der Testcases im Detail funktioniert.

**4. Erstellung von Testszenarien:**
Ist die Implementierung der Verifikationskomponenten vollendet, folgt die Erstellung von Testszenarien. Dazu werden oft mehrere verschiedene Randbedingungen sog. "Cornercases" und Standardbedingungen getestet. Zusätzlich können zufällige Eingangsfolgen getestet

werden, um schwer zu findende Fehler aufzudecken.

**5. Simulation:**
Nach dem Fertigstellen der Testszenarien, wurden die Eigenschaften der Simulatoren untersucht. Mit dem DUV SFIFO wurden insbesondere die Anzeigeeigenschaften des Simulators überprüft. Mit Hilfe des DUV Arbiter wurde überprüft, wie lange die UVM-Implementierungen brauchen, um den ersten Fehler zu finden. Dazu wurden im Arbiter bewusst mehrere Fehler eingebaut. Weiterhin wurden in beiden DUVs die UVM-Debugging Möglichkeiten im Simulator verglichen. Bei beiden DUVs wurde analysiert, ob es mit allen drei UVM-Implementierungen möglich ist, die o.g. Fehler zu finden.
Um dem Anwender einen Überblick über die Fehlerinformationen des Simulators geben zu können, wurden in die SFIFO UVM Umgebung gezielt Fehler eingebaut. Damit wurde geprüft, wie genau die Fehlerinformationen sind. Zusätzlich wurde die benötigte Simulationszeit für beide DUVs untersucht und mit der Kommandooption `-status` im Xcelium Simulator gemessen. Damit das Erstellen im Dateisystem nicht mit in die Simulationszeit mit einfließt, wurde nur die Zeit im Simulator `xmsim` gemessen.

**6. Durchführung einer Regression:**
Abschließend mussten alle Punkte des VPlans erfüllt sein. Hier wurden auch die Ergebnisse aus dem Simulator mit einbezogen. Dazu wurde in einem Regression-Tool (Auswerteeinheit) der Verifikationsplan mit den Testergebnissen verknüpft und überprüft, ob alle Punkte erreicht wurden. Wichtig bei der Regression sind die Anzeigemöglichkeiten von Tests, sowie Coverage und Assertions (Checks). Schlagen z.B. Testcases fehl oder ist noch nicht überall eine Coverage von 100% erreicht, wird das Verfahren zur Verifikationsplanung und UVM-Implementierung noch einmal von Anfang an durchlaufen. Hinzu kommt die Reparatur von identifizierten Fehlern im DUV. Damit alle Fehler schnell gefunden werden, wird das “Vier Augen Prinzip“ empfohlen. Das bedeutet, dass der Verifikationsprozess und das Verifikationsergebnis von einer zweiten Person überprüft wird. Da die meisten Regression-Tools die selben Simulatoren verwenden, spielt die Zeit hier eine geringe Rolle, sollte aber trotzdem Beachtung finden. Zudem wurde die Möglichkeit des Multiplexings (paralleles Simulieren mehrerer Testcases) bewertet.
Nach Abschluss aller Phasen sollte das Regression-Tool in allen Kategorien 100% anzeigen. Ist dies erreicht ist die Verifikation abgeschlossen und das DUV kann freigegeben werden.

Tabelle III
TOOL-UMGEBUNG.

| Tool | Version | Hersteller | Beschreibung |
|---|---|---|---|
| RHEL | 6 | Red Hat | Betriebssystem |
| vPlanner | 18.09-s004 | Cadence | Verifikationsplan |
| vManager | 18.09-s004 | Cadence | Regression-Tool |
| DVT-Eclipse | 19.1.40 | AMIQ EDA | Editor |
| Xcelium | 18.09.007 | Cadence | Simulator |
| GCC Compiler | 6.3.0 | GNU-Projekt | Compiler für u.a. C++ |
| SystemC Biblio. | 2.3.3 | Accellera | UVM-SystemC |
| UVM-Bibliothek | 1.0-beta2 | Accellera | UVM-SystemC |
| SCV Bibliothek | 2.0.1 | Accellera | SystemC Verifikation |
| FC4SC | 2.1.1 | AMIQ-Const. | Coverage Erzeugung |

### *E. Beschreibung der Entwicklungsumgebung*

Vor dem Vergleich der drei UVM-Implementierungen wird in diesem Kapitel die genutzte Softwareumgebung beschrieben. Als Betriebssystem wurde RHEL 6 verwendet. Der Verifikationsplan für alle drei UVM-Implementierungen wurde mit dem vPlanner der Firma Cadence erstellt. Für das Starten und Auswerten von Regressions wurde das Tool vManager der Fa. Cadence verwendet. Zum Editieren des Quellcodes fand das DVT-Eclipse 19.1.40 [9] Verwendung.
Für UVM-Specman *e* und UVM-SystemVerilog wurde als Simulator Xcelium 18.09.007 [10] der Firma Cadence (im Folgenden Xcelium) verwendet. Der Simulator stellt die entsprechenden UVM-Bibliotheken für die beiden UVM-Implementierungen bereit. Für UVM-SystemC wurde ein von der Firma Cadence abgewandelter GCC Compiler, Version 6.3.0, verwendet. Hierzu wurde die UVM-SystemC Bibliothek [11], sowie die UVM-Bibliothek [2], die SCV-Bibliothek [4] und die `FC4SC`-Bibliothek [6] eingebunden. Aufgrund der fehlenden Möglichkeit des gemeinsamen Simulierens von VHDL DUVs und UVM-SystemC im GCC Compiler, wurde der Xcelium Simulator verwendet. Der Simulator ermöglicht gemeinsames Simulieren von VHDL und UVM-SystemC.
Jedoch gibt es beim Xcelium Simulator einen entscheidenden Nachteil: Das abgewandelte UVM-SystemC implementiert keine Sequenzen [12, vgl. S. 10]. Deshalb muss die UVM-Bibliothek vom User eingebunden werden und die Coverage Bibliothek `FC4SC` muss wie die UVM-Bibliothek vom User initialisiert werden. Der User muss dadurch auf die UVM-Debugging Funktionen verzichten und er verliert damit die Möglichkeit die Coverage in Tools der Firma Cadence anzeigen zu können. Die Entscheidung für UVM-SystemVerilog und UVM-SystemC fiel auf die Tools der Fa. Cadence, da diese auch für UVM-Specman *e* verwendet werden. Die Hersteller Siemens EDA (ehemals Mentor Graphics) [13] und Synopsys Inc. (Synopsys) [14] bieten zwar Support für UVM-SystemVerilog und UVM-SystemC, dieser wurde jedoch in dieser Arbeit nicht überprüft. In Tabelle III sind zur Übersicht alle Tools und ihre genutzten Versionen aufgelistet.

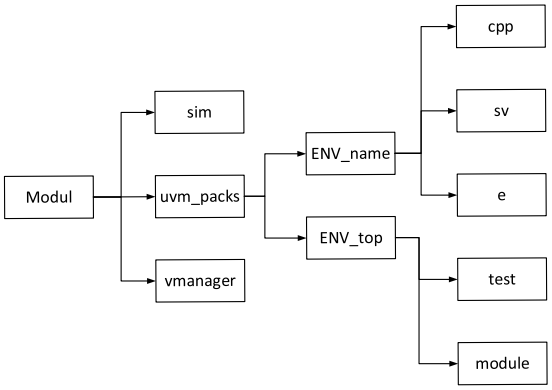



Abbildung 7. Ordnerstruktur

**Dateimanagement:**
Für ein erfolgreiches und systematisches Datenmanagement, wurden die Dateien nach folgendem Schema benannt: UVM-Specman *e* Dateien enden mit einem `.e`. UVM-SystemVerilog Dateien mit einem `.sv` und bei UVM-SystemC wurde `.cpp` als Endung für die Hauptdatei und `.h` für alle anderen Dateien verwendet.

Die Ordnerstruktur für jedes `Modul` wurde wie in Abbildung 7 dargestellt erstellt: In einem Ordner mit dem Namen `sim` wurden alle Skripte und Daten für die Simulation abgelegt. Daneben wurde ein Ordner `uvm_packs` zur Aufbewahrung aller Quellcode-Dateien angelegt. In diesem sind folgende Unterordner enthalten: Der Unterordner `ENV_name` enthält drei weitere Unterordner, die auf die UVM-Implementierung hinweisen: `e` für UVM-specman *e*, `sv` für UVM-SystemVerilog, `cpp` für UVM-SystemC. In diesen Unterordnern befinden sich dann die jeweiligen Quellcode Dateien. Im Unterordner `ENV_top` ist neben der UVM-Implementierung noch ein Unterordner mit den Testcases (`test`) und ein Unterordner `module` für das zu verifizierende DUV enthalten. In Abbildung 7 wurden aus Gründen der Übersichtlichkeit bei `ENV_top` die Ordner für die UVM-Implementierung weggelassen. Gibt es neben den beiden Environments weitere, so wurde für diese Environments analog zu dem Ordner `ENV_name` vorgegangen. Der Ordner `vmanager` wurde für die Regression Umgebung verwendet.

**Ausführen der Simulation und Regression:**
Die Ausführung der Simulation und Regression erfolgte für die jeweilige UVM-Implementierung skriptbasiert. Die Grundgerüste dieser Skripte sind für beide DUVs gleich und unterschieden sich nur durch die Übergabeparameter des Simulatoraufrufs.

### *F. Vergleich der drei UVM-Implementierungen*

Ist das Environment für die Verifikation aufgesetzt kann mit der Evaluierung entsprechend dem oben entwickelten Evaluierungsplan begonnen werden. Zunächst wurde der Plan auf das DUV SFIFO und im Anschluss daran auf das DUV Arbiter angewendet. Hierfür wurden die Schritte entsprechend dem Evaluierungsplan nacheinander ausgeführt. Um die drei UVM-Implementierungen miteinander vergleichen zu können, wurden diese anhand der in Kapitel III-B entwickelten 21 Kriterien zahlenmäßig bewertet. Hierfür wurden ganzzahlige Werte von 0 bis 10 vergeben wobei 0 die schlechteste und 10 die beste Bewertung ist. Im Folgenden werden zunächst die drei einzelnen UVM-Implementierungen bezüglich der 21 Kriterien diskutiert und zahlenmäßig bewertet:

**DUV-Language:**
Es besteht die Anforderung, dass alle drei UVM-Implementierungen Designs in den HDLs VHDL, Verilog und SystemC verifizieren können. Hierbei wurde auch die Abhängigkeit vom jeweiligen Simulator überprüft. Für Specman *e* wie auch für UVM-SystemVerilog wurden 10 Punkte vergeben, da beide UVM-Implementierungen die Möglichkeit bieten alle drei HDLs zu verifizieren. Bei der Verifikation mit UVM-SystemC stellte sich heraus, dass UVM-SystemC alle Hardwarebeschreibungssprachen unterstützt, jedoch gibt es eingeschränkte Funktionen bei DUVs in VHDL und Verilog. Dies kann durch Installation der vollständigen Bibliothek vom Nutzer selbst kompensiert werden. Dabei gehen allerdings die UVM-Debugging Funktionen verloren, wie sie in UVM-SystemVerilog und UVM-Specman *e* unterstützt wurden. Die anderen UVM-Implementierungen zeigen hier z.B. die Namen der einzelnen Signale an. Für UVM-SystemC wurden daher nur 5 Punkte vergeben.

**TLM-Kommunikation:**
Bezüglich der TLM-Kommunikation unterstützen alle drei UVM-Implementierungen die selben Kommunikationswege, jedoch unterscheiden sie sich im Aufruf der Funktionen. So kann bei UVM-Specman *e* ein Funktionsname festgelegt werden. Bei UVM-SystemVerilog kann der Funktionsname auch erweitert werden. Daher wurden für UVM-Specman *e* und UVM-SystemVerilog jeweils 10 Punkte vergeben. Bei UVM-SystemC ist die Funktion nur abhängig von dem übergebenden Parameter bzw. es kann durch weiteres TLM-Handling der Funktionsname verändert werden. Dafür muss eine neue Klasse instanziiert werden, der UVM-Subscriber. Das Einbinden des Subscribers benötigt zusätzliche Codezeilen im Skript, die bei UVM Specman *e* und UVM-SystemVerilog nicht benötigt werden. Jedoch kann durch Einsatz eines Templates dieser Aufwand reduziert werden. Zudem gibt es für UVM-SystemC nicht direkt die Möglichkeit im Scoreboard verschiedene TLM-Funktionen aufzurufen. Daher wurden für UVM-SystemC 8 Punkte vergeben.

**Konfigurierbarkeit:**
Beim Kriterium Konfigurierbarkeit wurden beim DUV SFIFO und beim DUV Arbiter keine Unterschiede zwischen den drei UVM-Implementierungen festgestellt. Beim DUV SFIFO gab es in allen UVM-Implementierungen die Möglichkeit die Agents passiv oder aktiv zu verwenden. Im DUV Arbiter war es zudem möglich den Namen des jeweiligen Master oder Slaves mittels der Konfiguration festzulegen. Aus diesem Grund erhielten alle drei UVM-Implementierungen die beste Bewertung (10 Punkte).

**Constrained Randomization:**
Bei der Randomisierung lassen sich in allen drei UVM-Implementierungen Constraints nutzen, jedoch können diese Constraints bei UVM-SystemVerilog nicht im Testcase festgelegt werden. Dafür müssen der SEQ-LIB Parameter übergeben werden. Dies erfordert mehr Aufwand (9 Punkte). Bei UVM-Specman *e* und UVM-SystemC ist die Übergabe aus dem Testcase möglich, jedoch ist die benötigte Zeit bis zur Fehleridentifikation deutlich größer (jeweils 9 Punkte). Voraussetzung dafür ist bei beiden Methoden, dass die Variablen in der SEQ-LIB deklariert werden. Bei der Einschränkung der Werte gibt es keine Unterschiede, dort können die 3 UVM-Implementierungen alle Einschränkungen vornehmen.

**Coverage:**
Für die Erzeugung von Coverage-Punkten stellen die UVM-Implementierungen UVM-Specman *e* und UVM-SystemVerilog Methoden zum Aufsammeln von Coverage zur Verfügung. Bei UVM-SystemC muss zusätzlich die `FC4SC` Bibliothek eingebunden werden. Dadurch werden Coverage Funktionen wie bei UVM-SystemVerilog erhalten und es lassen sich alle Coverage Punkte aufsammeln. Dazu können Funktionen wie z.B. `illegal bin` und `exclude bin` genutzt werden. UVM-SystemC unterstützt nicht direkt das Zusammenfassen mehrerer Variablen in einem Coverpunkt. Des Weiteren kann in UVM-SystemC von zwei verschiedenen Cross-Coverpunkten keine weitere Cross-Coverage gebildet werden (6 Punkte). Dies funktioniert nur bei UVM-Specman *e* und UVM-SystemVerilog (jeweils 10 Punkte).

**Checks:**
Checks können nur bei UVM-Specman *e* verwendet werden. Damit erlangte UVM-Specman *e* hier die höchste Punktzahl (10 Punkte). UVM-SystemVerilog und UVM-SystemC können Checks nur mit Hilfe von Assertions ausführen. Jedoch können sie in der Regression angezeigt werden. Die Nachbildung über Coverpunkte benötigt mehrere Zeilen im Skript und nimmt deshalb mehr Zeit in Anspruch (je 5 Punkte).

**Lines of Code:**
Dazu wurden die Lines of Codes der jeweiligen Simulatorskripte der jeweiligen DUVs verglichen. Es fiel auf, dass im SFIFO UVM-Specman *e* die wenigsten Zeilen benötigt. Beim Arbiter jedoch werden in UVM-SystemC die wenigsten Zeilen benötigt. Im SFIFO gibt es bei UVM-SystemC und UVM-SystemVerilog den Nachteil, dass die Bildung von Checks mehr Zeilen im Vergleich zu UVM-Specman *e* benötigt. Andererseits benötigte die Bildung der Coverage in UVM-SystemC mehr Zeilen als bei UVM-SystemVerilog bzw. UVM-Specman *e*. Beim Arbiter fiel auf, dass es möglich ist bei UVM-SystemC Arrays für die Listen zur verwenden. Dies spart bei der Deklaration und beim Aufruf und Speichern der Liste Zeilen ein. Zusammenfassend wurden UVM-Specman *e* und UVM-SystemC mit 8 Punkten, UVM-SystemVerilog mit 5 Punkten bewertet.

**UVM-Methoden:**
Im Bezug auf die UVM-Methoden können alle 3 Implementierungen alle Eigenschaften und Methoden unterstützen. Bei UVM-SystemC muss der Anwender die UVM-Bibliothek von Accellera selbst einbinden. Würde hingegen die UVM-SystemC Bibliothek der Firma Cadence verwendet müsste der Anwender auf Sequenzen verzichten. Daher wurden für alle UVM-Implementierungen die vollen 10 Punkte vergeben.

**Templates:**
Bei den Templates gibt es die umfangreichste Unterstützung bei UVM-Specman *e* (9 Punkte). Dabei kann zwischen Interface und Module ENV ausgewählt werden. Für UVM-SystemVerilog gibt es frei verfügbare Templates für die meistbenötigten Module, wie ENV, Agent, Driver usw.. Teilweise mussten diese angepasst werden, um dort einen besseren Nutzen zu erhalten (9 Punkte). Für UVM-SystemC wurden im Zuge dieser Arbeit Templates erstellt. Diese enthielten wie bei UVM-SystemVerilog nur die meistbenötigten Module (5 Punkte).

**Wiederverwendung (Re-use):**
Bei der Wiederverwendung von Komponenten bieten durch den UVM-Standard alle UVM-Implementierungen die selben Funktionen. Die meisten Funktionen können für alle drei UVM-Implementierungen genutzt werden. Einzelne Unterschiede ergaben sich bei Einbindung von neuen Signalen. Hier mussten bei UVM-Specman *e* die entsprechenden Signal Maps in mehreren Dateien angepasst werden (7 Punkte). Bei UVM-SystemVerilog und UVM-SystemC konnte diese Anpassung im Hauptmodul erfolgen (8 Punkte). Die Änderung im Item war hingegen bei allen UVM-Implementierungen gleich. Im Bezug auf die

Randomisierung fanden sich weitere Unterschiede. So musste bei UVM-SystemC die Randomisierung für das neue Signal in der SEQ-LIB hinzugefügt werden. Bei der Übernahme ganzer Module verhielten sich die UVM-Implementierungen wiederum gleich.

**Verification IPs:**
Bei den VIPs handelt es sich bei allen Herstellern bspw. um Interfaces wie z.B. I2C oder SPI. Für UVM-SystemVerilog wird ein breites Angebot von VIPs von den Herstellern Cadence [15], Siemens EDA [16] und Synopsys [17] bereitgestellt (10 Punkte). Für UVM-Specman *e* gibt es nur von der Firma Cadence Unterstützung mit einem breiten Abdeckungsbereich (6 Punkte). Dabei werden alle in UVM-SystemVerilog verfügbaren VIPs auch für UVM-Specman *e* angeboten. Für UVM-SystemC können die Lösungen von GreenSocs [18] verwendet werden. Zudem gibt es für einige VIPs von der Firma Cadence die Möglichkeit die VIPs aus UVM-SystemVerilog oder UVM-Specman *e* über UVM-ML einzubinden. Deshalb wurden für UVM-SystemC 4 Punkte vergeben.

**Weitergabe an den Kunden:**
Für die Weitergabe an Entwicklungspartner und Kunden gibt es verschiedene Voraussetzungen. Diese sind davon abhängig, welche Gegebenheiten dort vorliegen. Falls der Kunde über keine kommerziellen Tools verfügt, ist die Weitergabe bei UVM-SystemC am einfachsten möglich (8 Punkte). Bei UVM-SystemC kann die Verifikation unabhängig von kommerziellen Simulatoren durchgeführt werden. Jedoch ist der Funktionsumfang im Vergleich zu kommerziellen Tools eingeschränkt, z.B. darf das DUV nur in der Sprache UVM-SystemC implementiert sein. Für die Darstellung von Waveforms kann die UVM-SystemC Funktion `sc_trace` benutzt und mit einem Open-Source-Tool zur Anzeige gebracht werden. Als Open-Source-Tool kann beispielsweise `GTKWave` [19] und für die Coverage dann die `FC4SC` Bibliothek verwendet werden. Damit sich der Kunde nicht um die Einbindung der Tools bzw. Bibliotheken kümmern müss, können diese mit Hilfe eines Skripts installiert und initialisiert werden.
Für UVM-Specman *e* ist die Weitergabe nicht ohne weiteres möglich, da hier Lizenzen von Cadence benötigt werden. D.h. bei UVM-Specman *e* ist es von Vorteil, wenn der Kunde bereits UVM-Specman *e* verwendt. Verwendet der Kunde kein UVM-Specman *e* bzw. hat keine entsprechenden Lizenzen, könnte die Weitergabe nur mithilfe spezieller Zugänge zu den Servern des Unternehmens genutzt werden (3 Punkte). Für UVM-SystemVerilog wäre das Vorgehen ähnlich zu UVM-Specman *e*, jedoch gibt es hier durch die Entwicklung von Accellera theoretisch frei verfügbare Compiler wie z.B. Icarus Verilog [20]. Diese Option wurde allerdings im Rahmen dieser Arbeit nicht praktisch untersucht. Darüber hinaus ist es theoretisch möglich UVM-SystemVerilog mit anderen Simulatoren von bspw. Siemens EDA oder Synopsys zu simulieren (7 Punkte).

**EDA-Tool-Support:**
Der EDA-Tool-Support für die UVM-Implementierungen UVM-Specman *e* (9 Punkte) und UVM-SystemVerilog (10 Punkte) ist sehr ausgeprägt. Für UVM-SystemVerilog gibt es zusätzlich zu den Tools der Firma Cadence Support durch die Hersteller Synopsys und Siemens EDA. Des Weiteren liefert die Anzeige von Fehlermeldungen bei UVM-Specman *e* die präzisesten Informationen. Für UVM-SystemVerilog und UVM-SystemC gibt es theoretisch Support von Siemens EDA und Synopsys. Jedoch fehlen für UVM-SystemC die UVM- und die Coverage Funktionen (3 Punkte).

**Code Debugging im EDA-Tool:**
Bezüglich des Code Debugging im EDA-Tool lieferte im Rahmen dieser Untersuchung die Firma Cadence für UVM-Specman *e* die ausführlichsten Funktionen, hinzu kamen genaue Fehlermeldungen und die Möglichkeit die Konsole mit der Waveform zu verknüpfen (8 Punkte). Für UVM-SystemVerilog lieferte Cadence ebenfalls ausführliche Funktionen, wie bei UVM-Specman *e*. Einzig bei den Fehlermeldungen gab es teilweise ungenaue Hinweise (7 Punkte). Für UVM-SystemC wird der geringste Funktionsumfang bereitgestellt. Es gibt z.B. keine Möglichkeit die Konsole mit der Waveform zu verknüpfen. Für die Debugging-Funktionen von UVM-SystemC kann der User den GCC Compiler verwenden und er erhält wie bei UVM-Specman *e* und UVM-SystemVerilog ausführliche Debugging-Informationen (7 Punkte).

**UVM-Debugging:**
Aus der Verifikation des DUV SFIFO und des DUV Arbiter ergab sich, dass von den Tools der Firma Cadence die UVM-Debugging Funktionen nur für UVM-Specman *e* und UVM-SystemVerilog unterstützt werden (jeweils 10 Punkte). Dort war es möglich einzelne Phasen zu simulieren oder nach bestimmten Phasen die Simulation anzuhalten. Des Weiteren war es möglich die verschiedenen Sequenzen mit den jeweiligen Werten anzuzeigen. In den Tools der Firma Cadence gibt es derzeit keine Unterstützung für UVM-SystemC (0 Punkte).

**Anzeigen und Ausblenden von Konsolenausgaben:**
Für das Anzeigen und Ausblenden von Konsolenausgaben gibt es für jede UVM-Implementierung Möglichkeiten die Anzeigen zu filtern. Beispielsweise ist es möglich, nur Meldungen mit einer bestimmten

Tabelle IV
SIMULATIONSZEIT DES SFIFO.

| Durchlauf | Simulationszeit in s | | |
|---|---|---|---|
| | UVM *e* | UVM SV | UVM SC |
| 1 | 3,3 | 1,5 | 30,5 |
| 2 | 3,2 | 1,0 | 32,9 |
| 3 | 3,6 | 2,6 | 29,9 |
| Durchschnitt | 3,4 | 1,7 | 31,1 |

Tabelle V
SIMULATIONSZEIT DES ARBITER.

| Durchlauf | Simulationszeit in s | | |
|---|---|---|---|
| | UVM *e* | UVM SV | UVM SC |
| 1 | 4,2 | 3,0 | 58,8 |
| 2 | 2,7 | 2,3 | 40,6 |
| 3 | 2,6 | 2,7 | 59,0 |
| Durchschnitt | 3,2 | 2,7 | 52,8 |

Wertigkeit anzuzeigen. Zudem kann festgelegt werden, ob bei einem illegalen Coverpoint oder UVM-Error die Simulation anhalten oder weiterlaufen soll. Hier ist es auch möglich, bei illegalen Bins die Simulation zu stoppen oder weiterlaufen zu lassen (je 10 Punkte).

**Lizenzkosten:**
Lizenzkosten fallen für die Tools von Cadence an, deshalb wurde für UVM-Specman *e* 1 Punkt vergeben. UVM-SystemVerilog wurde in dieser Arbeit mit dem Tool Xcellium der Firma Cadence simuliert. Jedoch ist es theoretisch möglich UVM-SystemVerilog mit frei verfügbaren Tools zu simulieren. Aus diesem Grund wurden 3 Punkte vergeben. UVM-SystemC kann ohne kommerzielle Simulatoren simuliert werden, jedoch muss das DUV dann auch in UVM-SystemC implementiert sein. Wird UVM-SystemC bspw. mit dem Tool Xcellium der Firma Cadence simuliert, fallen Lizenzkosten an (6 Punkte). Alternativ kann UVM-SystemC mit einem frei verfügbaren Simulator verknüpft werden, um Designs in VHDL oder Verilog einzubinden. Jedoch erfordert dies Aufwand und wurde in dieser Arbeit nicht geprüft.

**Simulationszeit**
Die Simulationszeit des SFIFOs ist in Tabelle IV, die Simulationszeit des Arbiters in Tabelle V dargestellt. Bei beiden Modulen ist die Simulationszeit bei der UVM-Implementierung in UVM-SystemVerilog am kürzesten. Daher wurde UVM-SystemVerilog mit 10 Punkten bewertet. UVM-Specman-*e* benötigt bei beiden Modulen eine etwas höhere Simulationszeit als UVM-SystemVerilog und wird daher mit 7 Punkten bewertet. UVM-SystemC hat bei beiden Modulen die - verglichen mit UVM-SystemVerilog bis zum Faktor 20 - mit Abstand höchste Simulationszeit und erhält daher 1 Punkt.

**UVM-Multi-Language:**
Die Verifikation von DUVs, die in unterschiedlichen HDLs modelliert sind, wird von den Tools der Firma Cadence für alle drei UVM-Implementierungen unterstützt. Für UVM-SystemVerilog und UVM-Specman *e* kann dies ohne Einschränkungen verwendet werden. Daher wurden für diese beiden UVM-Implementierungen jeweils 10 Punkte vergeben. Für UVM-SystemC ist es auch möglich diese Methodik anzuwenden. Hierbei gibt es allerdings die Einschränkung, dass Xcelium keine Sequenzen unterstützt. Daher konnten diese nicht mit eingebunden werden. Aufgrund der fehlenden Unterstützung von Sequenzen in UVM-SystemC wurden nur 5 Punkte vergeben.

**Regression:**
Bei den Regression-Tools wurden die größten Unterschiede zwischen UVM-SystemC, UVM-SystemVerilog und UVM-Specman *e* festgestellt. Für alle drei UVM-Implementierungen kann angezeigt werden, welche Testcases ohne und welche mit einem Fehler durchgelaufen sind. Zudem ist es möglich für DUVs in VHDL oder Verilog die Coverage anzeigen zu lassen. Für die Coverage und Checks innerhalb der Verifikation ist es nur für UVM-Specman *e* und UVM-SystemVerilog möglich diese zur Anzeige zu bringen. Hierbei kann bei UVM-Specman *e* zusätzlich zwischen Coverage und Checks unterschieden werden. Für UVM-SystemC muss die separate `FC4SC` Bibliothek verwendet werden.
Beim Aufsetzen der Tool-Umgebung verhielten sich die drei UVM-Implementierung ähnlich. Für jede UVM-Implementierung wird ein Skript benötigt, welches das Tool initialisiert. Hierbei können mit einer zusätzlichen Datei vom Nutzer persönliche Einstellungen und Konfigurationen hinzugefügt werden. Bei UVM-Specman *e* und UVM-SystemVerilog werden jedoch die meisten Einstellungen vom Tool selbst übernommen. So müssen z.B. dem `xrun` nur die entsprechenden Befehle und Dateien mitgegeben werden. Bei UVM-SystemC muss jeder Simulator für die entsprechende Sprache selber initialisiert werden und zum Schluss müssen alle Simulationsergebnisse im Hauptsimulator verknüpft werden. Dazu sollte gewährleistet werden, dass alle Ergebnisse im selben Ordner gespeichert wurden bzw. zusätzlich die einzelnen Pfade angegeben werden. Bei der Regression gibt es die beste Unterstützung für UVM-Specman *e* (9 Punkte) und UVM-SystemVerilog (8 Punkte). Für UVM-SystemC fallen diese Regression-Funktionen weg, es kann lediglich angezeigt werden, ob ein Testcase fehlschlägt oder nicht (4 Punkte).

Tabelle VI
BEWERTUNGSTABELLE.

| Kategorie | Gesamt-punktzahl (G) | UVM-Specman *e* | | UVM-SystemVerilog | | UVM-SystemC | |
|---|---|---|---|---|---|---|---|
| | | Bewer-tung (B) | G·B | Bewer-tung (B) | G·B | Bewer-tung (B) | G·B |
| DUV Language | 18 | 10 | 180 | 10 | 180 | 5 | 90 |
| TLM-Kommunikation | 25 | 10 | 250 | 10 | 250 | 8 | 200 |
| Konfigurierbarkeit | 19 | 10 | 190 | 10 | 190 | 10 | 190 |
| Constrained Randomization | 32 | 9 | 288 | 9 | 288 | 9 | 288 |
| Coverage | 32 | 10 | 320 | 10 | 320 | 6 | 192 |
| Checks | 30 | 10 | 300 | 5 | 150 | 5 | 150 |
| LOC | 10 | 8 | 80 | 5 | 50 | 8 | 80 |
| UVM-Methoden | 16 | 10 | 160 | 10 | 160 | 10 | 160 |
| Templates | 8 | 9 | 72 | 9 | 72 | 5 | 40 |
| Wiederverwendung (Re-use) | 24 | 7 | 168 | 8 | 192 | 8 | 192 |
| VIPs | 7 | 6 | 42 | 10 | 70 | 4 | 28 |
| Weitergabe an den Kunden | 3 | 3 | 9 | 7 | 21 | 8 | 24 |
| EDA-Tool-Support | 21 | 9 | 189 | 10 | 210 | 3 | 63 |
| Code Debugging im EDA-Tool | 24 | 8 | 192 | 7 | 168 | 7 | 168 |
| UVM-Debugging | 21 | 10 | 210 | 10 | 210 | 0 | 0 |
| Anzeigen bzw. Ausbleden von Konsolenausgaben | 13 | 10 | 130 | 10 | 130 | 10 | 130 |
| Lizenzkosten | 40 | 1 | 40 | 3 | 120 | 6 | 240 |
| Simulationszeit | 23 | 7 | 161 | 10 | 230 | 1 | 23 |
| UVM-ML | 7 | 10 | 70 | 10 | 70 | 5 | 35 |
| Regression | 26 | 9 | 234 | 8 | 208 | 4 | 104 |
| Ein- und Ausblenden von Coverpunkten | 21 | 10 | 210 | 10 | 210 | 1 | 21 |
| Gesamtsumme | | | **3495** | | **3499** | | **2418** |

**Ein- und Ausblenden von Coverpunkten:** Mithilfe der Tools der Firma Cadence lassen sich nur bei UVM Specman *e* und UVM-SystemVerilog coverage- und check-Punkte anzeigen und im Nachhinein ausblenden (je 10 Punkte). Diese Funktion war beim Arbiter hilfreich, denn es wurden nicht alle Coverpunkte erreicht, da die Fehler nicht korrigiert wurden. Allerdings möchten User angezeigt bekommen, ob die Coverage erreicht wurde oder nicht. Bei UVM-SystemC ist dies nicht möglich, hier müssen die Coverage Punkte im Code ausgeblendet werden (1 Punkt). Das Ausblenden im Code führt dazu, dass die Regression neu gestartet werden muss. Sind die Module komplex, wird dazu viel Zeit benötigt. Alternativ kann die Anzeige in der `FC4SC` Bibliothek nach vier Kategorien gefiltert werden, die Gesamtwertung wird jedoch weiterhin von allen Coverpunkten gebildet. Bei kleineren DUVs wie dem SFIFO oder dem Arbiter ist dies ein geringer Aufwand. Für erfahrene Nutzer können jedoch in der XML-Datei die entsprechenden Coverpunkte ausgeblendet werden. Dadurch kann dieselbe Flexibilität erreicht werden, wie bei der Regression in Tools der Firma Cadence.

## IV. BEWERTUNG DER UVM-IMPLEMENTIERUNGEN

Nach Abschluss der Bewertung aller 21 Kriterien wurde die jeweilige Bewertung (B) jedes einzelnen Kriteriums entsprechend Kapitel III-F mit der Gesamtpunktzahl (G) jedes Kriteriums aus Tabelle II für jede der drei UVM-Implementierungen multipliziert. Die Gesamtpunktzahl jedes Kriteriums aus Tabelle II ist in Tabelle VI in der zweiten Spalte dargestellt. In den Spalten 3, 5 und 7 findet sich die jeweilige individuelle Bewertung des Kriterums, wie sie in Kapitel III-F erläutert wurde. Die Produkte aus Gesamtpunktzahl und individueller Bewertung (G·B) sind in den Spalten 4, 6 und 8 dargestellt. Durch Aufsummieren der Produkte aus Gesamtpunktzahl und indiviueller Bewertung ergibt sich die Gesamtsumme pro UVM-Implementierung. Diese ist in der untersten Zeile von Tabelle VI dargestellt und gibt Aufschluss darüber welche UVM-Implementierung am besten geeignet ist. UVM-SystemVerilog erreicht mit 3499 Punkten die höchste Punktzahl. UVM-Specman *e* hat mit 3495 Punkten die zweithöchste Punktzahl und UVM-SystemC mit 2418 Punkten die geringste Punktzahl.
UVM-SystemVerilog und UVM-Specman *e* sind in vielen Kategorien ähnlich bewertet, daher beträgt der Unterschied zwischen den beiden UVM-Implementierungen nur 4 Punkte. Gemeinsamkeiten gibt es beispielsweise bei der DUV-Language oder der TLM-Kommunikation. Die größten Unterschiede liegen in den Lizenzkosten, den Checks und den VIPs.
UVM-SystemC verliert in den Kategorien EDA-Tool-Support, UVM-Debugging und Ein- und Ausblenden von Coverpunkten deutlich gegenüber den anderen beiden UVM-Implementierungen. Dies bedeutet, dass der Unterschied hauptsächlich an den Tools der EDA-

Hersteller liegt. Die größten Vorteile erreicht UVM-SystemC bei der Konfigurierbarkeit und den Lizenzkosten. Zudem liegt UVM-SystemC bei den Lines of Code gleichauf mit UVM-Specman *e*. Bei der Wiederverwendung gibt es zudem kaum Unterschiede zwischen UVM-SystemVerilog und UVM-SystemC.

## V. Zusammenfassung

Im Rahmen dieser Arbeit wurde gezeigt, dass UVM-SystemC aktuell für die Schaltungsverifikation anwendbar ist. UVM-SystemC unterstützt alle UVM-Methoden und es ist möglich Designfehler zu identifizieren. Es kann eine Coverage gebildet und durch das Seed Management können verschiedene Seeds verwendet werden. Weiterhin können neben DUVs in UVM-SystemC auch DUVs verifiziert werden, die in den HDLs VHDL oder Verilog modelliert sind. Durch die Unterstützung von UVM-ML ist es möglich, Designs, die aus Blöcken in unterschiedlichen HDLs bestehen, zusammen zu verifizieren.

Die größten Vorteile der Schaltungsverifikation mit UVM-SystemC liegen in den nicht vorhandenen Lizenzkosten. UVM-SystemC kann kostenlos genutzt werden. Das erleichtert auch die Zusammenarbeit mit Entwicklungspartnern, Kunden oder externen Designhäusern, insbesondere wenn diese keine kommerziellen CAD-Tools lizensiert haben.

Bei den Kriterien Konfigurierbarkeit, Constrained Randomization, den benötigten Lines of Code für das Verifikationsskript, UVM-Methoden und Re-Use ist UVM-SystemC den beiden etablierten UVM-Implementierungen durchaus ebenbürtig, teilweise sogar überlegen.

Jedoch gibt es für UVM-SystemC Einschränkungen gegenüber den beiden UVM-Implementierungen UVM-Specmen *e* und UVM-SystemVerilog.

Großes Verbesserungspotential besteht beim eingeschränkten Support der Fa. Cadence für UVM-SystemC. Der Wegfall der UVM-Debugging Möglichkeiten, ist hier der größte Nachteil. Hierbei wäre es wünschenswert, wenn die Firma Cadence die vollständige UVM-Bibliothek bereitstellen würde. Der Anwender kann die UVM-Bibliothek selber einbinden, muss sich jedoch bei der Ausführung der Simulation um viele Einstellungen selbst kümmern.

Eine weitere Möglichkeit zur Verbesserung liegt beim Simulator. Hier sollte die Möglichkeit bestehen zwischen der Konsole und der Waveform zu wechseln. Für das Anzeigen der Coverage wurde bis zum Ende dieser Arbeit keine Möglichkeit gefunden, sie in der Regression anzeigen zu können. Jedoch ist es möglich die Coverage mithilfe der `FC4SC` Bibilothek zur Anzeige zu bringen. Dabei muss der User jedoch Einschränkungen beim Ein- und Ausblenden von Coverpunkten hinnehmen.

## Literaturverzeichnis

[1] Accellera Systems Initiative, ”Universal Verification Methodology (UVM) 1.2 User’s Guide”, https://www.accellera.org/downloads/standards/uvm, 21.10.2022.

[2] Accellera Systems Initiative, ”UVM-SystemC Library 1.0-beta2”, https://www.accellera.org/downloads/drafts-review, 21.10.2022.

[3] Cadence Design Systems, ”Universal Verification Methodology (UVM) e User Guide”, 2014.

[4] Accellera Systems Initiative, ”SystemC Verification 2.0.1: SystemC Verification Library”, https://www.accellera.org/downloads/standards/systemc, 21.10.2022.

[5] Group of Computer Architecture of the University of Bremen FB3, ”CRAVE: Constrained Random Verification Environment”, http://www.informatik.uni-bremen.de/agra/systemc-verification/crave.html, 21.10.2022.

[6] Dragos Dospinescu, ”FC4SC” https://www.amiq.com/consulting/2018/11/15/new-release-of-the-functional-coverage-for-systemc-library/, 21.10.2022.

[7] Cadence Design Systems, ”SystemC SCV/CVE Library Reference: Product Version 14.1”, 2014.

[8] AlDanial, ”cloc”, https://github.com/AlDanial/cloc/releases/tag/1.84, 21.10.2022.

[9] AMIQ EDA, ”DVT Eclipse IDE”, https://dvteclipse.com/products/dvt-eclipse-ide, 21.10.2022.

[10] Cadence Design Systems, ”Xcelium: 18.09.007,” https://www.cadence.com/ko_KR/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html, 14.03.2020.

[11] Accellera Systems Initiative, ”SystemC 2.3.3 (Includes TLM): Core SystemC Language and Examples”, https://www.accellera.org/downloads/standards/systemc, 21.10.2022.

[12] Cadence Design Systems, ”UVM-SC Library Reference: UVM Version 1.0”, 2011.

[13] Mentor, a. Siemens Business, ”Questa Verification Solution”, https://eda.sw.siemens.com/en-US/ic/questa/ 21.10.2022.

[14] Synopsys Inc., ”Verdi”, https://www.synopsys.com/verification/debug/verdi.html, 21.10.2022.

[15] Cadence Design Systems, ”Verification IP”, https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html, 21.10.2022.

[16] Siemens EDA, ”Verification IP”, https://eda.sw.siemens.com/en-US/ic/questa/simulation/verification-management/, 21.10.2022.

[17] Synopsys Inc., ”Verification IP”, https://www.synopsys.com/verification/verification-ip.html, 21.10.2022.

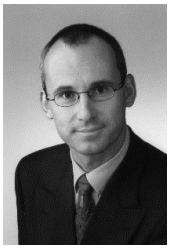
[18] Mark Burton, ”GreenSocs”, https://www.machineware.de/#qemu, 21.10.2022.

[19] Udi Finkelstein, ”GTKWave,” http://gtkwave.sourceforge.net/, 21.10.2022.

[20] Stephen Williams, ”Icarus Verilog”, http://iverilog.icarus.com/home, 21.10.2022.

**Moritz Kupke** schloss sein Bachelorstudium der Elektrotechnik und Informationstechnik mit der Vertiefungsrichtung Mikroelektronik an der Hochschule Düsseldorf im Jahr 2017 ab. Seine Masterstudium Mikroelektronik - wiederum an der Hochschule Düsseldorf - schloss er im Jahr 2020 ab. Parallel zu seinem Masterstudium war er von 2017 bis 2019 wissenschaftlicher Mitarbeiter an der Hochschule Düsseldorf. Seit Mitte 2020 ist er als ASIC-Entwicklungsingenieur bei der Firma Bosch GmbH in Dresden beschäftigt.

**Stephan Gerth** schloss sein Studium der Informatik an der Brandenburgischen Technischen Universität Cottbus (BTU) im Jahr 2008 ab. Im Anschluss arbeitete er im Fraunhofer Institut IIS/EAS als wissenschaftlicher Mitarbeiter am Entwurf von ESL-Modellen und der dazugehörigen Entwurfs- und Verifikationsmethodik und übernahm später die Gruppenleitung. Seit 2018 ist er für die Robert Bosch GmbH am Standort Dresden tätig und leitet die Verifikation in verschiedenen Projekten.



**Bernhard Rieß** schloss sein Studium der Elektrotechnik und Informationstechnik an der Technischen Universität München im Jahr 1992 ab. Anschließend war er wissenschaftlicher Mitarbeiter am Lehrstuhl für Rechnergestütztes Entwerfen an der TU München und wurde dort 1996 zum Dr.-Ing. promoviert. Seit Anfang 1997 war er als Entwicklungsingenieur bei der Infineon Technologies AG in München/Neubiberg tätig. 2012 wurde er von der Hochschule Düsseldorf als Professor für das Lehrgebiet Mikroelektronik berufen.